Data Entered)

AD-A221 449

ION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

12. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Ada Compiler Validation Summary Report: DRUPP ATLAS ELEKTRONIK GMBH, DRUPP ATLAS ELEKTRONIK Ada Compiler VVME 1.81, VAX 6310 (Host) to KRUPP ATLAS ELEKTRONIK GMBH MPR 2300 (Target), 891124I1.10235

5. TYPE OF REPORT & PERIOD COVERED

24 Nov. 1989 to 24 Nov. 1990

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

IABG,
Ottobrunn, Federal Republic of Germany.

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

IABG,
Ottobrunn, Federal Republic of Germany.

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

IABG,
Ottobrunn, Federal Republic of Germany.

15. SECURITY CLASS (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 if different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DRUPP ATLAS ELETRONIK Ada Compiler VVME 1.81, Ottobrunn, West Germany, VAX 6310 under VMS, Version 5.1 (Host) to KRUPP ATLAS ELEKTRONIK GMBH MPR 2300 under EOS 2300, Version 1.4 (Target), 891124I1.10235

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: AVF-IABG-034

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #891124I1.10235
KRUPP ATLAS ELEKTRONIK GMBH
KRUPP ATLAS ELEKTRONIK Ada Compiler VVME 1.81
VAX 6310 Host
KRUPP ATLAS ELEKTRONIK GMBH MPR 2300 target

Completion of On-Site Testing:
24th November 1989

Prepared By:
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: KRUPP ATLAS ELEKTRONIK Ada Compiler VVME 1.81

Certificate Number: #891124I1.10235

Host: VAX 6310 under VMS, Version 5.1

Target: KRUPP ATLAS ELEKTRONIK GMBH
MPR 2300 under EOS 2300, Version 1.4

Testing Completed Friday 24th November 1989 Using ACVC 1.10

This report has been reviewed and is approved.

IABG mbH, Abt SZT
Dr S. Heilbrunner
Einsteinstr 20
D8012 Ottobrunn
West Germany

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC 20301

ORDER OF PAGINATION CORRECT AND NO MISSING PAGES(PAGES INSERTED AND NOT RENUMBERED) per Michele Key, ADA Info. Clearing House, c/o ITT Research Inst., 4600 Forbes Blvd., Lanham, MD 20706

TELECON 4/27/90 VG

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability, (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed Friday 24th November 1989 at KRUPP ATLAS ELEKTRONIK GmbH, Bremen, Germany.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from

IABG mbH, Abt. SZT
Einsteinstr 20
D8012 Ottobrunn

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: KRUPP ATLAS ELEKTRONIK Ada Compiler VVME Version 1.81

ACVC Version: 1.10

Certificate Number: #891124I1.10235

Host Computer:

Machine:	VAX 6310
Operating System:	VMS Version 5.1
Memory Size:	32 MB

Target Computer:

Machine:	KRUPP ATLAS ELEKTRONIK GMBH MPR 2300
Operating System:	EOS 2300 Version 1.4
Memory Size:	4 MB

Communications Network: Ethernet

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types SHORT_INTEGER, SHORT_FLOAT, and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

4) No exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..N (14 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..N (14 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..N (14 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

This implementation evaluates the 'LENGTH of each constrained array subtype during elaboration of the type declaration. This causes the declaration of a constrained array subtype with more than INTEGER'LAST (which is equal to SYSTEM.MAX_INT for this implementation) components to raise CONSTRAINT_ERROR. However, the optimisation mechanism of this implementation suppresses the evaluation of 'LENGTH if no object of the array type is declared depending on whether the bounds of the array are static, the visibility of the array type, and the presence of local subprograms. These general remarks apply to points (1) to (5), and (8).

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception if the bounds of the array are static. (See test C36003A.)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components if the bounds of the array are not static and if the subprogram declaring the array type contains no local subprograms. (See test C36202A.)

3) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components if the bounds of the array are not static and if the subprogram declaring the array type contains a local subprogram. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. However this implementation raises USE_ERROR upon creation of a file for unconstrained array types.(See tests AE2101H, EE2401D, and EE2401G.)

3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 481 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 159 executable tests that use floating-point precision exceeding that supported by the implementation and 238 tests containing file operations not supported by the implementation. Modifications to the code, processing, or grading for 14 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1132	1852	17	16	46	3192
Inapplicable	0	6	463	0	12	0	481
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	202	591	566	245	172	99	161	331	137	36	252	325	75	3192
N/A	11	58	114	3	0	0	5	1	0	0	0	44	245	481
Wdrn	0	1	0	0	0	0	0	2	0	0	1	35	5	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 481 tests were inapplicable for the reasons indicated:

a. The following 159 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C241130..Y (11 tests)	C357050..Y (11 tests)
C357060..Y (11 tests)	C357070..Y (11 tests)
C357080..Y (11 tests)	C358020..Z (12 tests)
C452410..Y (11 tests)	C453210..Y (11 tests)
C454210..Y (11 tests)	C455210..Z (12 tests)
C455240..Z (12 tests)	C456210..Z (12 tests)
C456410..Y (11 tests)	C460120..Z (12 tests)

b. C34007P and C34007S are expected to raise CONSTRAINT_ERROR. This implementation optimizes the code at compile time on lines 205 and 221 respectively, thus avoiding the operation which would raise CONSTRAINT_ERROR and so no exception is raised.

c. C41401A is expected to raise CONSTRAINT_ERROR for the evaluation of certain attributes, however this implementation derives the values from the subtypes of the prefix at compile time as allowed by 11.6 (7) LRM. Therefore elaboration of the prefix is not involved and CONSTRAINT_ERROR is not raised.

d. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

e. C45531M..P (4 tests) and C45532M..P (4 tests) are inapplicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

f. C47004A is expected to raise CONSTRAINT_ERROR whilst evaluating the comparison on line 51, but this compiler evaluates the result without invoking the basic operation qualification (as allowed by 11.6 (7) LRM) which would raise CONSTRAINT_ERROR and so no exception is raised.

g. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

h. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

i. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

j. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

l. CD1009C, CD2A41A, CD2A41B, CD2A41E and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I and CD2A61J are not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

n. CD2A71A..D (4 tests), CD2A72A..D (4 tests), CD2A74A..D (4 tests) and CD2A75A..D (4 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for record types.

o. CD2A84B..I (8 tests), CD2A84K and CD2A84L are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.

p. The following 238 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)
CE2102K	CE2102N..Y (12 tests)
CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)
CE2107A..H (8 tests)	CE2107L
CE2108A..B (2 tests)	CE2108C..H (6 tests)
CE2109A..C (3 tests)	CE2110A..D (4 tests)
CE2111A..I (9 tests)	CE2115A..B (2 tests)
CE2201A..C (3 tests)	CE2201F..N (9 tests)
CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A
CE3102A..B (2 tests)	EE3102C
CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)
CE3107B	CE3108A..B (2 tests)
CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)

CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A
CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)
CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A..B (2 tests)	CE3412A
EE3412C	CE3413A
CE3413C	CE3602A..D (4 tests)
CE3603A	CE3604A..B (2 tests)
CE3605A..E (5 tests)	CE3606A..B (2 tests)
CE3704A..F (6 tests)	CE3704M..O (3 tests)
CE3706D	CE3706F..G (2 tests)
CE3804A..P (16 tests)	CE3805A..B (2 tests)
CE3806A..B (2 tests)	CE3806D..E (2 tests)
CE3806G..H (2 tests)	CE3905A..C (3 tests)
CE3905L	CE3906A..C (3 tests)
CE3906E..F (2 tests)	

These tests were not processed because their inapplicability can be deduced from the result of other tests.

q. Tests CE2103A, CE2103B and CE3107A raise USE_ERROR upon create for Sequential, Direct and Text IO.

r. Tests EE2201D, EE2201E, EE2401D and EE2401G raise USE_ERROR upon create.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 14 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B51001A	B91001H	BA1101E	BC2001D	BC2001E	BC3204B
BC3205B	BC3205D				

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the KRUPP ATLAS ELEKTRONIK Ada Compiler VVME Version 1.81 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the KRUPP ATLAS ELEKTRONIK Ada Compiler VVME Version 1.81 using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	VAX 6310
Host operating system:	VMS Version 5.1
Target computer:	KRUPP ATLAS ELEKTRONIK GMBH MPR 2300
Target operating system:	EOS 2300 Version 1.4
Compiler:	KRUPP ATLAS ELEKTRONIK Ada Compiler VVME Version 1.81

The host and target computers were linked via Ethernet.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 6310, then all executable images were transferred to the target via Ethernet and run. Results were printed

from the host computer.

The compiler was tested using command scripts provided by KRUPP ATLAS ELEKTRONIK and reviewed by the validation team. Executable tests were compiled using the command

```
$@ADA:COMPILE <test name> OPTIONS = LIST => OFF
                                    OPTIMIZER => ON
                                    INLINE => ON
                                    COPY_SOURCE => OFF
```

and linked with the command

```
$@ADA:LINK <test name> <test name>.EXE COMPLETE => ON
                                       DEBUG => OFF
                                       SELECT => ON
                                       LOADMAP => OFF.
```

Chapter B tests were compiled with the full listing option

```
$@ADA:COMPILE <test name> OPTIONS = LIST => ON
                                    LIST = <listfile name>.
```

A full description of compiler and linker options is given in Apendix E.

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at KRUPP ATLAS ELEKTRONIK, Bremen West-Germany, and was completed on Friday 24th November 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

KRUPP ATLAS ELEKTRONIK has submitted the following Declaration of Conformance concerning the KRUPP ATLAS ELEKTRONIK Ada Compiler VVME Version 1.81:

DECLARATION OF CONFORMANCE

Compiler Implementor: KRUPP ATLAS ELEKTRONIK GMBH
Ada Validation Facility: IABG MBH
Ada Compiler Validation Capability (ACVC) Version: 1.10.

BASE CONFIGURATION

Base Compiler Name:	KRUPP ATLAS ELEKTRONIK Ada Compiler VVME 1.81
Base Compiler Version	1.81
Host Architecture ISA:	VAX 6310 under VMS 5.1
Target Architecture ISA:	KRUPP ATLAS ELEKTRONIK GMBH MPR 2300 under EOS 2300 1.4

IMPLEMENTOR'S DECLARATION

We, the undersigned, representing KRUPP ATLAS ELEKTRONIK GMBH have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. We declare that KRUPP ATLAS ELEKTRONIK GMBH is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

KRUPP ATLAS ELEKTRONIK GMBH

- Brötje -

- Barg -

OWNER'S DECLARATION

We, the undersigned, representing KRUPP ATLAS ELEKTRONIK GMBH, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. We declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

KRUPP ATLAS ELEKTRONIK GMBH

- Brötje-

- Barg-

Bremen, November 21st, 1989

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the KRUPP ATLAS ELEKTRONIK Ada Compiler VVME Version 1.81, as described in this Appendix, are provided by KRUPP ATLAS ELEKTRONIK GmbH. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Chapter 7 (Representation Clauses and Implementation-Dependent Features) and chapter 8 (Input - Output) of the KRUPP ATLAS ELEKTRONIK Ada System User Manual are also listed in this Appendix. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

     type INTEGER is range - 2_147_483_648 .. 2_147_483_647;
     type SHORT_INTEGER is range - 32_768 .. 32_767;

     type FLOAT is digits 15 range
        - 16#0.FFFF_FFFF_FFFF_F8#E32 .. 16#0.FFFF_FFFF_FFFF_F8#E32;
     type SHORT_FLOAT is digits 6 range
        - 16#0.FFFF_F8#E32 .. 16#0.FFFF_F8#E32;
     type LONG_FLOAT is digits 18 range
        - 16#0.FFFF_FFFF_FFFF_FFFC#E256 ..
          16#0.FFFF_FFFF_FFFF_FFFC#E256;

     type DURATION is delta 2#1.0#E-7 range
        - 16_777_216.0 .. 16_777_216.0;

   ...

end STANDARD;
```

9 Appendix F

This is the Appendix F required in [Ada], in which all implementation-dependent characteristics of an Ada implementation are described.

9.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

9.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here. All the other pragmas listed in Appendix B of [Ada] are implemented and have the effect described there.

CONTROLLED

has no effect.

INLINE

Inline expansion of subprograms is supported with following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for the languages Assembler and Meta. For each Ada subprogram for which

PRAGMA interface (<language> , <ada_name>);

is specified, a routine implementing the body of the subprogram <ada_name> must be provided, written in the specified language. The name of the routine, which implements the subprogram <ada_name>, should be specified using the **PRAGMA EXTERNAL_NAME**, otherwise the compiler will generate an enternal name that leads to an unsolved reference during linking.

The subprogram <ada_name> specified in the **PRAGMA INTERFACE** may be a function or a procedure. For the interface (META,...) there are following conventions. For parameter passing the programmer has to specify a record in Ada with USE- and AT-clauses. The META-program takes a pointer to the physical structure of the record as its one and only parameter. There are also restrictions in the use of the META-language.

- META-Tasking actions (e.g. META-Parallel) are not allowed.
- The User-semaphores from 0 to 1023 are used for the runtime-system and not available for the programmer. Their use of them may lead to an erronous program.
- The use of the Space-Monitor (SPAMON) of the MOS is restricted, too. For Ada-Data-Space the first segmented and the first not segmented subspace available through SPAMON are already used.

Example:

Ada:

```
..
TYPE parameter IS
  RECORD
     length : natural;
     addr   : system.address;
  END RECORD;


FOR parameter USE
 RECORD AT MOD 4;
    length AT 0 RANGE 0 .. 31;
    addr   AT 4 RANGE 0 .. 31;
 END RECORD;


PROCEDURE put_line ( x : IN parameter );
PRAGMA INTERFACE (META, put_line);
PRAGMA EXTERNAL NAME ("P_L", put_line);
..
```

META:

```
PROC: P_L (PB POINTER_TO_PARAMETER);

TYPE PARAMETER STRUCT /
                LENGTH  LONG;
                ADDRESS POINTER;
```

For the interface (ASSEMBLER,...) the parameter passing is similiar to interface (META,...). You specify a record in Ada with the correct structure of the parameters. You can reference them in the assembler program via the A1-register, which is known as parameter base. The following conventions have to be obayed by the assembler programmer:

- First of all save all used registers by a MOVEM.L instruction onto the user stack.
- Take care of the stackpointer.
- Reference parameters via A1.
- In the end restore all saved registers.
- Return to the calling routine with an RTS-instruction.

Example:

Ada:

```
..
TYPE mos_time IS
      RECORD
        year        : integer;
        month       : integer;
        day         : integer;
        seconds     : integer;
      END RECORD;

FOR mos_time USE
       RECORD
         year       AT 0*4 RANGE 0 .. 31;
         month      AT 1*4 RANGE 0 .. 31;
         day        AT 2*4 RANGE 0 .. 31;
         seconds    AT 3*4 RANGE 0 .. 31;
       END RECORD;

FOR mos_time'size USE 16*system.storage_unit;

FUNCTION mos_clock RETURN mos_time;
   PRAGMA INTERFACE (Assembler, mos_clock);
   PRAGMA EXTERNAL_NAME ("_TSKCLCK", mos_clock);
..
```

Assembler:

```
          XDEF      _TSKCLCK
_TSKCLCK  EQU       *

* Save registers

          MOVEM.L   A0-A7/D0-D7,-(A7)

* Take Time from MOS

          MOVE.L    A1,-(A7)
          MOVEQ.L   #5, D0
          TRAP      #6
          MOVEA.L   (A7)+,A1

* Parameters via A1-registers

          BFEXTU    D1{16:7},D0
          ADD.L     #1900,D0
          MOVE.L    D0,([A1],0)
          BFEXTU    D1{23:4},D0
          MOVE.L    D0,([A1],4)
          BFEXTU    D1{27:5},D0
          MOVE.L    D0,([A1],8)
          TMULS.L   #DU_SMALL,D2
          ADD.L     #99,D2
          TDIVS.L   #TENMILLI,D2
          MOVE.L    D2,([A1],12)

* restore registers
```

```
MOVEM.L  (A7)+,A0-A7/D0-D7
RTS
```

The KRUPP ATLAS ELEKTRONIK Ada Compiler does not provide checking the observance of the procedure calling standard. If it is violated the call of the system routine will be erroneous.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect.

PACK

see User Manual

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of the subtype **PRIORITY**, and second, the effect on scheduling of not giving this pragma for a task or main program. The range of subtype **PRIORITY** is 0 .. 63 as declared in the predefined library **PACKAGE system** and the effect on scheduling of leaving the priority of a task or main program undefined by not giving **PRAGMA priority** for it is the same as if **PRAGMA priority** 63 had been given (i.e. the task has the highest priority). Moreover, in this implementation the **PACKAGE system** must be named by a with clause of a compilation unit if the predefined **PRAGMA priority** is used within that unit.

SHARED

is supported.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see for the implementation-defined PRAGMA suppress_all.

SYSTEM_NAME

has no effect.

9.1.2 Implementation-Defined Pragmas

SQUEEZE

is supported

SUPPRESS_ALL

causes all the run_time checks except ELEBORATION_CHEK to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

EXTERNAL_NAME (<string>, <ada_name>)

<ada_name> specifies the name of a subprogram, <string> must be a string literal. The string has a maximum length of 8 characters. It denotes the external name that the compiler place with the entry point of the specified sub-program. The suprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which preceds immediately. This pragma will be used in connection with **PRAGMA interface (Meta)** or **interface (Assembler)**

RESIDENT (<ada_name>)

this pragma causes that no assigment of a value to the object <ada_name> will be eliminated by the optimizer of the **KRUPP ATLAS ELEKTRONIK Ada Compiler**. The following code sequence demonstrates the intended usage of the pragma:

```
..
   x : integer;
   a : SYSTEM.address;
   PROCEDURE do_something (a : SYSTEM.address);

   ..
   BEGIN
     x := 5;
     a := x'ADDRESS;
     do_something (a);  -- a.ALL will be read in the body
                        -- of do_something
     x := 6;
..
```

If this code sequence is compiled by the **KRUPP ATLAS ELEKTRONIK Ada Compiler** with the option

```
OPTIMIZER=>ON
```

the statement x := 5; will be eliminated because from the point of view of the optimizer the value of x is not used before the next assignment to x. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of x. This pragma can be applied to all those kinds of objects for which the address clause is supported

9.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this chapter.

9.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in [Ada]. We note here only the implementation-dependent aspects.

ADDRESS

The value delivered by this attribute applied to an object is the address of the storage unit where this object starts. For any other entity this attribute is not supported and will return the value system.address_zero.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows: If a length specification (STORAGE_SIZE) has been given for that type (static collection), the attribute delivers that specified value. In case of a dynamic collection, i.e. no length specification by STORAGE_SIZE given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed. If the collection manager is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.
The value delivered by this attribute applied to a task type or task object is as follows:
If a length specification (STORAGE_SIZE) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned elsewhere.

9.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

9.3 Specification of the Package SYSTEM

The PACKAGE system of [Ada,$13.7)) is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS
   TYPE designated_by_address IS LIMITED PRIVATE;
   TYPE address IS ACCESS designated_by_address;
   FOR address'size USE 32;
   For address'storage_size USE 0;
    -- Logically, the type address is defined by:
    --   TYPE Address IS PRIVATE;
    -- However, in this case no representation specification
    -- can be given for record components of type system.address.
    -- The storage size specification assures that any attempt
    -- to create an address value with an allocator raises
    -- STORAGE ERROR
   address_zero : CONSTANT address := NULL;
   FUNCTION "+" (left : address; right : integer) RETURN address;
      PRAGMA _built_in (address_plus_integer, "+");
   FUNCTION "+" (left : integer; right : address) RETURN address;
      PRAGMA _built_in (integer_plus_address, "+");
   FUNCTION "-" (left : address; right : integer) RETURN address;
      PRAGMA _built_in (address_minus_integer, "-");
   FUNCTION "-" (left : address; right : address) RETURN integer;
      PRAGMA _built_in (address_minus_address, "-");
   SUBTYPE external_address IS STRING;
    -- External addresses use hexadecimal notation with characters
    -- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
    --     "7FFFFFFF"
    --     "80000000"
    --     "8" represents the same address as "00000008"
   FUNCTION convert_address (addr:external_address) RETURN
            address;
    -- convert_address raises CONSTRAINT_ERROR if the external
            address
      -- addr is the empty string, contains characters other than
      -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
      -- value cannot be represented with 32 bits.

  FUNCTION convert_address (addr : address) RETURN
            external_address;
    -- The resulting external address consists of exactly 8
            characters
      -- '0'..'9', 'A'..'F'.

   TYPE name IS (motorola_68020_kae);
   system_name  : CONSTANT name := motorola_68020_kae;
   storage_unit : CONSTANT := 8;
   memory_size  : CONSTANT := 2 ** 31;
   min_int      : CONSTANT := - 2 ** 31;
   max_int      : CONSTANT := 2 ** 31 - 1;
   max_digits   : CONSTANT := 18;
   max_mantissa : CONSTANT := 31;
   fine_delta   : CONSTANT := 2.0 ** (-31);
   tick         : CONSTANT := 0.01;
   SUBTYPE priority IS integer RANGE 0 .. 63;

   non_ada_error : EXCEPTION RENAMES _no_ada_error;
    -- non_ada_error is raised if some event occurs which does not
    -- correspond to any situation covered by Ada, e.g.:
    --     illegal instruction encountered
    --     error during address translation
    --     illegal address
```

```
   TYPE exception_id IS NEW integer;

  no_exception_id      : CONSTANT exception_id := 0;
   -- Coding of the predefined exceptions:
  constraint_error_id : CONSTANT exception_id :=  16#0002_0000;
  numeric_error_id    : CONSTANT exception_id :=  16#0002_0001;
  program_error_id    : CONSTANT exception_id :=  16#0002_0002;
  storage_error_id    : CONSTANT exception_id :=  16#0002_0003;
  tasking_error_id    : CONSTANT exception_id :=  16#0002_0004;
PRIVATE
   TYPE designated_by_address IS NEW integer;
END system;
```

9.4 Restrictions on Representation Clauses

See $$7.2-7.5 of this manual.

9.5 Conventions for Implementation-Generated Names

There are no implementation-generated names denoting implementation-dependent components [Ada,$13.4].

9.6 Expressions in Address Clauses

Address clauses [Ada,$13.5] are supported for objects except for task objects. The object starts at the given address.

9.7 Restrictions on Unchecked Conversions

9.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in Chapter 14 of [Ada] are reported in Chapter 8 of this manual.

7 Representation Clauses and Implementation-Dependent Features

In this chapter we follow the section numbering of Chapter 13 of [Ada] and provide notes for the use of the features described in each section.

7.1 Representation Clauses

PRAGMA pack

As stipulated in [Ada,$13.1], this pragma may be given for a record or array type. It causes the compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type **PRAGMA pack** has no affect on the mapping of the component type. For all other component types the compiler will try to choose a more compact representation for the component type. All components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of system.storage_unit. Thus, **PRAGMA pack** does not effect packing down to the bit level (for this see **PRAGMA squeeze**).

PRAGMA squeeze

This is an implementation-defined pragma which takes the same argument as the predefined language **PRAGMA pack** and is allowed at the same positions. It causes the compiler to select a representation for the argument type that needs minimal storage space (packing down to the bit level). For components whose type is an array or record type **PRAGMA squeeze** has no affect on the mapping of the component type. For all other component types the compiler will try to choose a more compact representation for the component type. The components of a squeezed data structure will not in general start at storage unit boundaries.

7.2 Length Clauses

SIZE

for all integer, fixed point and enumeration types the value must be <= 32; for short_float types the value must be = 32 (this is the amount of storage which is associated with these types anyway);
for float and long_float types the value must be = 64 (this is the amount of storage which is associated with these types anyway).
for access types the value must be = 32 (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the compiler responds with a RESTRICTION error message in the compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage spaceneeded to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 4K bytes if no length clause is given (cf. Chapter 6). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (maximum length is 256 kByte for each task virtuell.)

SMALL

there is no implementation-dependent restriction. Any specification for SMALL that is allowed by the LRM can be given. In particular those values for SMALL are also supported which are not a power of two.

7.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; that is the type integer defined in **PACKAGE standard**.

7.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the compiler responds with a RESTRICTION error message in the compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1 (resp. 2 or 4) the starting address of an object will be a multiple of 1 (resp. 2 or 4) * storage unit size.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifiing a bigger range than needed.) Violation of this restriction will produce a RESTRICTION error message.

There are implementation-dependent components generated to hold the size of the record object if the corresponding record type includes variant parts or to hold the offset of a record component (relative to this generated component) if the size of the record component is dynamic. But there are no implementation-generated names (cf. [Ada,$13.4(8)]) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

7.5 Address Clauses

Address clauses are supported for objects declared by an object declaration. If an address clause is given for a subprogram, package, task unit or single entry, the compiler responds with a RESTRICTION error message in the compiler listing.

7.6 Change of Representation

The implementation places no restrictions on changes of representation.

7.7 The Package SYSTEM

See $9.3. The pragmas system_name, storage_unit and memory_size have no effect.

7.7.1 System-Dependent Named Numbers

See $9.3.

7.7.2 Representation Attributes

These are all implemented.

7.7.3 Representation Attributes of Real Types

These are all implemented.

7.8 Machine Code Insertions

A PACKAGE machine_code is not provided and machine code insertions are not supported.

7.9 Interface to Other Languages

The PRAGMA interface is provided for META and Assembler language of MPR2300 subprogram that obeys the calling conventions of the MOS procedure calling standard. It is described in §9.1.1 of this manual.

7.10 Unchecked Programming

7.10.1 Unchecked Storage Deallocation

The implementation does not support unchecked storage deallocation. (The generic PROCEDURE unchecked_deallocation is provided, but the only effect of calling an instantiation of this procedure with an object X as actual parameter is

```
X := NULL;
```

i.e. no storage is reclaimed.)

7.10.2 Unchecked Type Conversions

The implementation does not support unchecked type conversions.

8 Input-Output

In this chapter we follow the section numbering of Chapter 14 of [Ada] and provide notes for the use of the features described in each section.

8.1 External Files and File Objects

File-management is not supported, cause embedded systems don't need any File-I/O. Any use of DIRECT_IO and SEQUENTIAL_IO raises the exception USE_ERROR.

Files associated with terminal devices (which is only legal for text files) may be opened with an arbitrary mode at the same time and associated with the same terminal device.

8.2 Text Input/Output

Text input/output is only implemented for standard I/O and devices. So in the following "Text Files" means an output to a device. Text files are represented as sequential files with variable record format. One line is represented as a sequence of one or more records; all records except for the last one have a length of exactly MAX_RECORD_SIZE and a continuation marker ("SPACE" = 16#20#) at the last position. A line of length MAX_RECORD_SIZE is represented by one record of this length. The end of a record which is shorter than MAX_RECORD_SIZE or which has length exactly MAX_RECORD_SIZE and does not have a continuation marker as its last character is taken as a line terminator.

The value MAX_RECORD_SIZE is fixed to 255 byte.

Line terminators, page terminators and file terminators are not represented explicitly on the external file. The effect of these terminators is determined by respective Standard-I/O-functions of the MOS 2300 operating system. A record of length zero is assumed to precede a page terminator if the record before the page terminator is another page terminator or a record of length MAX_RECORD_SIZE with a continuation marker at the last position; this implies that a page terminator is preceded by a line terminator in all cases.

The end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last record of the file.

8.2.1 File Management

The association of an external file with MOS 2300 filename convention is only supported if the MOS 2300 filesystem is present. Otherwise the MOS 2300-error-messages will trigger the predefined Ada exception USE_ERROR.

8.2.2 The NAME and FORM Parameters

The name parameter string must be a MOS 2300 device specification string and must not contain wild cards. The FUNCTION name will return a device specification string which is the device name of the device opened or created.

The syntax of the form parameter string is defined by:

```
form_parameter ::= [ form_specification { , form_specification } ]

 form_specification ::=  keyword [ => value ]
```

```
keyword ::= identifier

value   ::= identifier | string_literal | numeric_literal
```

For identifier, numeric_literal, string_literal see [Ada,Appendix E]. Only an integer literal is allowed as numeric_literal (see [Ada,$2.4]).

DEVICE
In the following, the form specifications which are allowed for all files are described. If the keyword "DEVICE" is used as form-specification, then the internal file is associated with a MOS 2300 device (D.name). This form parameter is only allowed in an open statement. The Device name must denote a legal external MOS 2300 device. An external MOS 2300 device may be opened in mode "in" or "out".

8.3.3 Default Input and Output Files

The standard input (resp. output) file is associated with the terminal device of MOS2300 ("*").

The name string for the standard files is assumed to be :

```
standard_input  :  NAME => Terminaldevicename
standard_output :  NAME => Terminaldevicename
```

8.2.4 Implementation-Defined Types

The implementation-dependent types count and field defined in the package specification of **TEXT_IO** have the following upper bounds :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)

FIELD'LAST = 255
```

8.3 Exceptions in Input-Output

For each of NAME_ERROR, USE_ERROR, DEVICE_ERROR and DATA_ERROR we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package IO_EXCEPTIONS can be raised are as described in [Ada,$14.4].

NAME_ERROR

is never raised. Instead of this exception the exception USE_ERROR is raised whenever an error occurred during an operation of the underlying MOS operating system.

USE_ERROR

- if an attempt is made to increase the total number of open files (including the two standard files) to more than 32;

- whenever an error occurred during an operation of the underlying MOS2300 operating system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons;

- if the function name is applied to a temporary file;

- if the characteristics of the external file are not appropriate for the file type; for example, if the record size of a file with fixed-length records does not correspond to the size of the element type of a DIRECT_IO or SEQUENTIAL_IO file. In general it is only guaranteed that a file which is created by an Ada program may be reopened by another program if the file types and the form strings are the same;

- if two or more (internal) files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), and an attempt is made to open one of these files with mode other than in_file. However, files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device;

- if a given form parameter string does not have the correct syntax or if a condition on an individual form specification described in $8.2.2 is not fulfilled;

. if an attempt is made to open or create a sequential or direct file.

DEVICE_ERROR

is never raised. Instead of this exception the exception USE_ERROR is raised whenever an error occurred during an operation of the underlying MOS operating system.

DATA_ERROR

. the conditions under which DATA_ERROR is raised in the package TEXT_IO are laid down in [Ada].

8.4 Low Level Input-Output

not implemented

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	254 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	254 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	127 * 'A' & '3' & 127 * 'A'

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	127 * 'A' & '4' & 127 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	252 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	250 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 127 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 127 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	235 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2_147_483_648
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

TEST PARAMETERS

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	MOTOROLA_68020_KAE
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE_AVAILABLE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	16_777_217.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	63
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	MUCH_TOO_LONG1_TX
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	MUCH_TOO_LONG2_TX

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	0.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-16_777_217.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	18
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

TEST PARAMETERS

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 250 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 248 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 253 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	MOTOROLA_68020_KAE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2_147_483_648
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	MOTOROLA_68020_KAE
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. -- 63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.-CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

This appendix contains information concerning the compilation and linkage commands used within the command scripts for this validation.

3 Compiling, Linking and Executing a Program

3.1 Overview

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can be all on the same file. One unit, a parameterless procedure, acts as main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting executable files can then be transfered to the MPR2300 and executed by a *ADA command.

$3.2 and $3.4 describe in detail how to call the compiler and the linker. The information the compiler produces and outputs in the compiler listing is explained in $3.2.1. Further on in $3.3 the completer, which is called to generate code for instances of generic units, is described.

$3.5 and $3.6 describe how to transfer to and excecute a program on MPR2300.

$3.7 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.

Finally, the log of a sample session is given in $3.8.

3.2 Starting the Compiler

To start the **KRUPP ATLAS ELEKTRONIK Ada Compiler**, call the command

```
@VMK:COMPILE  <source> [LIBRARY=<directory>] -
                       [OPTIONS=<string>]    -
                       [LIST=<filespec>]
```

The input file for the compiler is <source>. If the file type of <source> is not specified, <source>.ADA is assumed. The maximum length of lines in <source> is 80; longer lines are cut and an error is reported.

<directory> is the name of the program library; [.ADALIB] is assumed if this parameter is not specified. The library must exist (see §2.2 for information on program library management).

The listing file is created in the default directory with the file name of <source> and the file type .LIS if no file specification <filespec> is given by the parameter LIST. Otherwise, the directory and file name are determined by the file specification <filespec>. If no full file specification is given, missing components are determined as described above (i.e. the default directory is used if no directory is specified, the file name of <source> if no file name is specified and the file type .LIS if the file type is missing). See §3.6 for information about the listing.

Options for the compiler can be specified by using the parameter OPTIONS; they have an effect only for the current compilation. <string> must have the syntax:

"[option {, option}]"

where blanks are allowed following and preceding lexical elements within the string.

The compiler accepts the following options:

```
(LIST           => ON/OFF )
(OPTIMIZER      => ON/OFF )
(INLINE         => ON/OFF )
(COPY_SOURCE    => ON/OFF )
(SUPPRESS_ALL             )
(SYMBOLIC_CODE            )
```

The options **LIST** and **SUPPRESS_ALL** have the same effect as the corresponding pragmas would have at the beginning of the source (see [Ada,Appendix B] and $9.1.2 of this Manual).

No optimizations like constant folding, dead code elimination or structural simplifications are done if **OPTIMIZER => OFF** is specified.

Inline expansion of subprograms which are specified by **PRAGMA inline** (cf. $9.1.1) in the Ada source can be suppressed generally by giving the option **INLINE => OFF**. The value **ON** will cause inline expansion of the respective subprograms.

COPY_SOURCE => ON causes the compiler to copy the source file <source> into the program library. This option is already implemented for a debugger. The debugger of the **KRUPP ATLAS ELEKTRONIK Ada System** will be implemented later. It works on this copy (cf. $2.2.7) instead of on the original file.
The generation of a symbolic code listing can be achieved by the option **SYMBOLIC_CODE** and produces a file in the default directory with the name of <source> and file type **.SYM**.

The source file may contain a sequence of compilation units (cf. Chapter 10.1 of [Ada]). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see $3.2.1). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The compiler delivers the status code **WARNING** on termination (see [VAX/VMS, DCL Dictionary, command EXIT]) if one of the compilation units contained errors. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

3.3 The Completer

The compiler does not generate code for instances of generic bodies. Since this must be done before a program using such instances can be executed, the COMPLETER tool must be used to complete such units. This is done implicitly when PRELINK is called.

It is also possible to call the completer explicitly by

```
@VMK:COMPLETE  <ada_name> [LIBRARY=<directory>] -
                          [OPTIONS=<string>]    -
                          [LIST=<filespec>]
```

<ada_name> must be the name of the library unit, not the filename. All library units that are needed by that unit (cf. [Ada,$10.5]) are completed, if possible, and so are their subunits, the subunits of those subunits and so on. The meaning of the parameters **LIBRARY** and **LIST** corresponds to that of the **COMPILE** command (cf. $3.2). Options apply to all units that are completed; the following ones are accepted (cf. $3.2):

```
(OPTIMIZER => ON/OFF)
(INLINE    => ON/OFF)
(SUPPRESS_ALL       )
(SYMBOLIC_CODE      )
```

The completer delivers the status code WARNING on termination (see [VAX/VMS, DCL Dictionary, command EXIT]) if it detected some error. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

In this case a listing file is created that contains the error messages (cf. $3.2.1). If no file specification <filespec> is given by the parameter **LIST**, the listing file is created in the default directory with file name **COMPLETE** and the file type .LIS; otherwise, the directory and file name are determined by the file specification <filespec>. If no full file specification is given, missing components are determined as described above (i.e. the default directory is used if no directory is specified, the filename

COMPLETE if no filename is specified and the file type **.LIS** if the file type is missing).

3.4 The Linker

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

To link a program, call the command

```
@VMK:LINK   <ada_name>  <filename>  [LIBRARY=<directory>] -
                                    [OPTIONS=<string>] -
                                    [LIST=<filespec>] -
                                    [COMPLETE=ON/OFF] -
                                    [DEBUG=ON/OFF] -
                                    [SELECT=ON/OFF] -
                                    [EXTERNAL=<string>] -
                                    [LOADMAP=ON/OFF]
```

<ada_name> is the name of the library procedure, not the filename which acts as the main program.

<filename> is the name of the file which is to contain the executable code after linking. If no filetype is specified, **.KAX** is assumed. KAX is the short form for KRUPP ATLAS ELEKTRONIK ADA EXECUTABLE.

<directory> is the name of the program library which contains the main program; [**.ADALIB**] is assumed if this parameter is not specified.

The **COMPLETE** parameter specifies whether the program is to be completed before it is linked; default is **ON**. If the completer is called, the parameters **LIBRARY**, **OPTIONS** and **LIST** are passed to it (cf. §3.3).

The **DEBUG** parameter is already implemented for a later version of the debugger. It specifies whether information for the **KRUPP ATLAS ELEKTRONIK** Debugger are to be generated; default is **ON**.

SELECT=ON causes only those object modules which are needed during program execution to be linked together. In the case of **OFF** all imported compilation units are linked together; the default is **ON**. **LOADMAP=ON** causes the generation of a file on the default directory with the name of <source> and the file type **.MAP**.

The following steps are performed during linking. First the Completer is called, unless suppressed by **COMPLETE=OFF**, to complete the bodies of instances. Then the Pre-Linker is executed; it determines the compilation units that have to be linked together and a valid elaboration order. A code sequence to perform the elaboration is generated.

The prelinker of the **KRUPP ATLAS ELEKTRONIK Ada System** delivers the status code WARNING on termination (see [VAX/VMS, DCL Dictionary, command EXIT]) if one of the above mentioned steps failed (e.g. if one of the completed units contained errors, if any compilation unit cannot be found in the program library or if no valid eleboration order can be determined because of incorrect usage of the **PRAGMA elaborate**). A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.